МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №2
по курсу «Программирование графических процессоров»

Обработка изображений на GPU. Фильтры.

Выполнил: И.Т. Батяновский
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Цель работы: Научиться использовать GPU для обработки изображений.
Использование текстурной памяти.

Вариант 5. Метод Робертса.

Программное и аппаратное обеспечение

Device: GeForce GTX 970
Размер глобальной памяти: 4294967296
Размер константной памяти : 65536
Размер разделяемой памяти: 49152
Регистров на блок: 65536
Максимум потоков на блок: 1024
Количество мультипроцессоров : 13
OS: Windows 10
Редактор: CLion

Метод решения

Процесс основан на простом перемещении маски фильтра от точки к точке изображения; в каждой точке отклик фильтра вычисляется с использованием перекрестного градиентного оператора Робертса. Поскольку при будет произведено много обращений к памяти, следует воспользоваться текстурной памятью, которая работает быстрее благодаря кэшированию. Результат обработки каждого пикселя будет записываться в выходной массив.

Описание программы

texture<uchar4, 2, cudaReadModeElementType> texRef;
Создается текстурная ссылка в качестве глобального объекта, после чего выделил память для массива на девайсе, скопировал в нее массив-изображение и сделал бинд ссылки с массивом:

```
cudaArray *arr;
cudaChannelFormatDesc ch = cudaCreateChannelDesc<uchar4>();
cudaMallocArray(&arr, &ch, n, m);
cudaMemcpyToArray(arr, 0, 0, h_data, size, cudaMemcpyHostToDevice);

texRef.addressMode[0] = cudaAddressModeClamp;
texRef.addressMode[1] = cudaAddressModeClamp;
texRef.channelDesc = ch;
texRef.filterMode = cudaFilterModePoint;
texRef.normalized = false;

cudaBindTextureToArray(texRef, arr, ch);
```

Далее вызывается ядро

```
RobertsMethod<<<dim3(512, 512), dim3(32, 32)>>>(dev_data, n, m);
```

В самом ядре вычисляется градиент и проверяется на превышает ли он значение 255(проверка на переполнение uchar).

```
if (x < width && y < height) {
    Gx = colourParser(tex2D(texRef, x + 1, y)) - colourParser(tex2D(texRef, x, y + 1)),
    Gy = colourParser(tex2D(texRef, x, y)) - colourParser(tex2D(texRef, x + 1, y + 1));

}

if (x < width && y < height)
{
    int grad = sqrtf(Gx * Gx + Gy * Gy);
    temp = grad < 256? grad: 255;
}
```

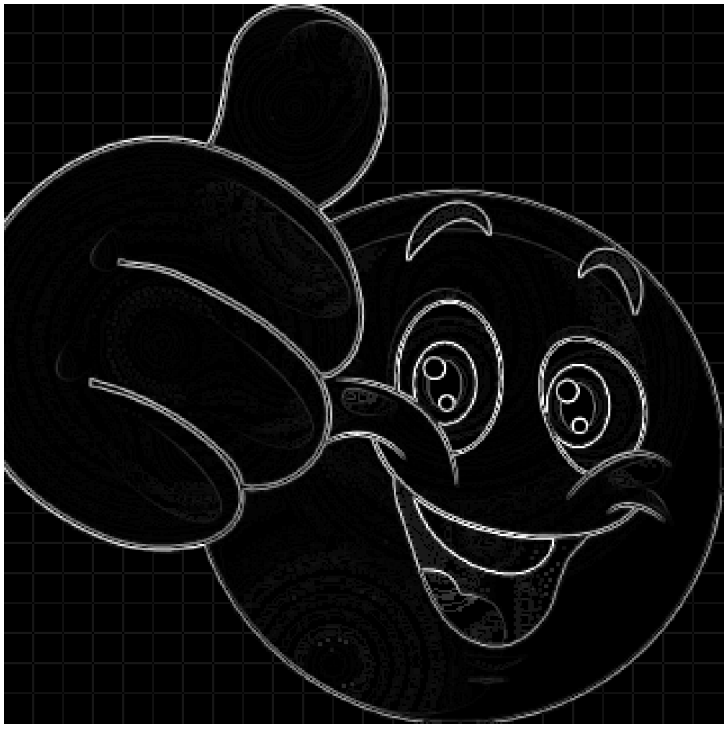
Далее сохраняем полученное значение.

```
if (x < width && y < height)
{
    output[y * width + x].x = temp;
    output[y * width + x].y = temp;
    output[y * width + x].z = temp;
    output[y * width + x].w = 0;
```

}

Результаты

После чего я применил конвертер в заданный формат. После чего я применил к ним свою программу, и конвертировал обратно.



Я посмотрел как зависит время работы на этих изображениях от количества запущенных потоков:

Threads, size	Test 1, ms	Test 2, ms
32 * 32	0.023	0.032
64 * 64	0.002	0.003
256 * 256	0.006	0.002
4096 * 4096	0.002	0.003
8192 * 8192	0.003	0.002

Отсюда видно, что начиная с некоторого количества потоков, производительность работы на GPU не возрастает. При этом этот порог для разных размеров изображений разный, что довольно объяснимо, поскольку при большем размере данных требуется большее количество нитей для оптимального распараллеливания алгоритма.

Выводы

В ходе выполнения работы возникла трудность в том, как расположить данные в текстурной памяти так, чтобы влезть в ограничения, а также было совсем не очевидно, что градиент может принять значение больше 255, что приводило к неверным результатам из-за переполнения.

Алгоритмы хорошо распараллеливается, что делает эффективным их использование на графических процессорах. Реализованный мной алгоритм широко применяется при обработке изображений, поскольку позволяет четко выделить контуры, что бывает полезно в задачах машинного обучения.